Projet : interpréteur YoctoForth
Semestre impair 2022-2023

1 Introduction

Le projet qui est décrit ci-après doit être réalisé de manière individuelle.
Le travail que vous allez produire sera évalué :

- à partir d'un dépôt sur Moodle que vous devrez effectuer au plus tard le mardi 3 janvier 2023 à midi (n'attendez pas le dernier moment...) ; quelle que soit la raison, si le dépôt est vide le 3 janvier 2023 à midi alors vous aurez 0 ; il est donc conseillé de déposer votre projet avant, même s'il n'est pas fini ; mais la version déposée doit pouvoir être compilée sans erreur ;
- à partir d'un oral qui se déroulera :
  - le mercredi 4 janvier 2023 à 18h pour les groupes A11, A12, A21 et A22 ;
  - le vendredi 6 janvier 2023 à 15h45 pour le groupe A31 ;

1.1 Barème

Le projet comporte quatre niveaux de difficulté croissante, numérotés de 1 à 4. La réalisation complète et exacte de chaque niveau donne la possibilité d'obtenir une certaine note maximale, comme cela est indiqué dans le tableau suivant :

Niveau	Note maximale
1	12
2	14
3	17
4	20

Pour commencer un niveau, il faut avoir complètement réalisé le niveau précédent. Par exemple, si vous n'avez pas terminé le niveau 2 et si vous réalisez une partie du niveau 3, vous serez tout de même noté sur 14. Un niveau est considéré comme réalisé quand toutes ses fonctionnalités sont programmées et que l'interpréteur s'exécute sans erreur.
Attention : si le projet que vous avez déposé sur Moodle ne peut pas être compilé sans erreur en utilisant la commande `make`, vous aurez automatiquement la note 0.

1.2 Dépôt sur Moodle

Vous déposerez sur Moodle un seul fichier archive au format ZIP qui doit être produite en suivant les directives suivantes.

1. Créez un répertoire dont le nom doit suivre le modèle :
   NOM-PRENOM-NUMETU
   où :
   - NOM, PRENOM et NUMETU doivent être remplacés, respectivement et dans cet ordre, par votre nom de famille en majuscules, votre prénom en majuscules et votre numéro d'étudiant (8 chiffres) ;
   - n'apparaît aucune lettre accentuée ;
   - n'apparaît aucun espace ni apostrophe, les éventuels espaces et apostrophes dans votre nom ou votre prénom devant être remplacés par des tirets - (signe moins) ;
2. Placez dans ce répertoire :

- les fichiers sources des modules de votre projet, c'est-à-dire les fichiers d'extensions `.h` et `.c`, y compris `memoire.h` et `memoire.c` ;
- un fichier de nom `yforth.c` contenant la fonction principale (`main`) de votre projet ;
- un fichier de description des dépendances de nom `Makefile` permettant, grâce à la commande `make`, de produire l'exécutable `yforth` ;
- un fichier de texte (obtenu avec un simple éditeur de texte) de nom `niveau-n.txt` où vous remplacerez `n` par le numéro du niveau que vous avez réalisé ; ce fichier pourra être vide ou contenir d'éventuelles informations destinées à expliquer les choix que vous aurez faits, surtout pour le niveau 4.

Vos fichiers sources doivent être indentés et contenir des commentaires permettant de bien comprendre votre travail. Aucun autre fichier (objets, exécutable, etc.) ne doit être présent dans le répertoire.

3. Créez une archive au format ZIP de ce répertoire ; cette archive doit porter le même nom que le répertoire auquel est ajoutée l'extension `.zip` ; pour produire cette archive, dans le terminal, vous pouvez vous placer dans le répertoire parent du répertoire contenant vos fichiers sources et utiliser la commande :
`zip␣NOM-PRENOM-NUMETU.zip␣NOM-PRENOM-NUMETU/*`

Respectez bien toutes ces consignes car des tests et des vérifications automatiques seront effectués sur votre projet.

1.3 Oral

Au moment de l'oral, vous devrez être prêt à :
- faire fonctionner votre programme ;
- répondre aux questions de l'enseignant ;
- effectuer des tests ou des modifications demandées par l'enseignant.

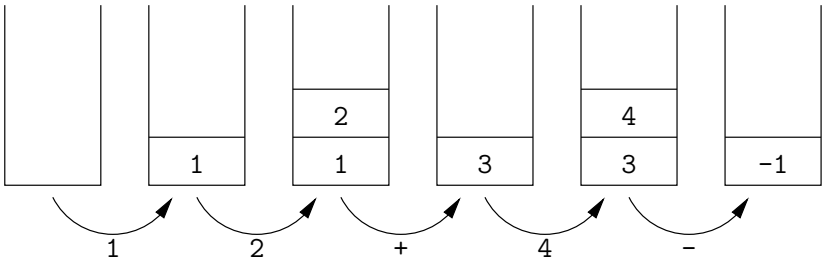
2 Objectif

Le but de ce projet est d'écrire un interpréteur pour un langage dérivé du langage Forth. Pour des soucis de simplicité, l'interpréteur YoctoForth ne fonctionnera pas en mode interactif, mais il lira un programme écrit dans un fichier, le chargera en mémoire et l'exécutera.
Attention : votre projet devra prendre en compte uniquement la syntaxe du langage YoctoForth et avoir le comportement décrit dans ce sujet et non pas celui d'autres dérivés du langage Forth que vous pourriez être amenés à connaître.

3 Principes généraux du langage YoctoForth

Il s'agit d'un langage à pile utilisant la notation *post-fixée* (parfois aussi appelée *polonaise inverse*). Dans cette famille de langages, on peut citer par exemple PostScript, Forth ou RPL, le langage des calculatrices HP. Avec cette notation, les opérandes sont donnés avant les opérateurs. Par exemple, l'opération en notation *infixée* `1+2-4` se note
`1␣2␣+␣4␣-` où ␣ désigne le caractère espace.

Ces langages utilisent explicitement des piles, dont une *pile de données* dans laquelle sont empilés les opérandes et les résultats des opérations. Par exemple, si l'on suppose que la pile de données est initialement vide et que l'on exécute `1␣2␣+␣4␣-`, la pile de données va évoluer de la manière suivante :



4 Description du langage YoctoForth

4.1 Programme YoctoForth

Un programme YoctoForth est stocké dans un fichier de texte et est constitué de mots (instructions ou valeurs entières) séparés par au moins un caractère séparateur (espace, tabulation ou « nouvelle ligne »). Les minuscules et majuscules ne sont pas différenciées. Les données numériques manipulées sont des entiers. Un mot (instruction) est composé d'une suite de caractères quelconques autres que des séparateurs.

4.2 Convention de description des mots du langage

Pour décrire les mots du langage, on utilisera dans ce sujet la convention suivante :

<mot> `(` *<pile avant>* `--` *<pile après>* `,` *<explication>* `)`

où *<mot>* désigne le mot décrit, *<pile avant>* est une représentation des éléments concernés situés au sommet de la pile avant l'exécution du mot (les paramètres), *<pile après>* est une représentation des éléments empilés lors de l'exécution du mot (les résultats) et *<explication>* est une courte description du rôle du mot. En règle générale, lors de l'exécution d'un mot, ses paramètres sont dépilés et le ou les résultats sont empilés.
Remarque : dans les représentations des éléments de la pile, l'élément le plus à droite est celui qui est situé au sommet.

4.3 Mots de base

4.3.1 Manipulations de la pile de données

`.S ( -- , affiche à l'écran le contenu de la pile en la laissant intacte )`
`. ( n -- , ôte l'entier au sommet de la pile et l'affiche à l'écran )`
`DUP ( n -- n n , duplique l'entier au sommet de la pile )`
`DROP ( n -- , supprime l'entier au sommet de la pile )`
`SWAP ( a b -- b a , permute les deux entiers au sommet de la pile )`
`OVER ( a b -- a b a , copie l'avant dernier entier au sommet de la pile )`

Remarque : avec le mot `.S`, les informations affichées sont : le nombre d'éléments présents dans la pile placé entre les caractères `<` et `>` suivi des éléments de la pile en terminant par l'élément se trouvant à son sommet. Le caractère `>` ainsi que chaque élément de la pile doivent être suivis d'un caractère espace.
Exemples : dans tous les exemples qui suivent, on suppose que la pile est initialement vide.

Les mots :	provoquent l'affichage de :
12␣4␣-1␣.s	<3>␣12␣4␣-1␣
12␣1␣+␣.	13
3172␣dup␣.s	<2>␣3172␣3172␣
1␣2␣drop␣.s	<1>␣1␣
1␣2␣.s␣swap␣.s	<2>␣1␣2␣<2>␣2␣1␣
1␣2␣.s␣over␣.s	<2>␣1␣2␣<3>␣1␣2␣1␣

4.3.2 Opérations arithmétiques

Les opérations arithmétiques à deux opérandes fonctionnent selon le schéma suivant : le second opérande est l'entier au sommet de la pile et le premier est celui situé juste en dessous ; ces deux entiers sont ôtés de la pile et le résultat de l'opération est empilé.

`+ ( a b -- a+b , addition )`
`- ( a b -- a-b , soustraction )`
`* ( a b -- a*b , multiplication )`
`/ ( a b -- a/b , quotient de la division entière )`
`MOD ( a b -- a modulo b , reste de la division entière )`

Exemples :

Les mots :	provoquent l'affichage de :
3␣4␣+␣.	7
3␣4␣-␣.	-1
3␣4␣*␣.	12
7␣2␣/␣.	3
7␣2␣mod␣.	1

4.3.3 Comparateurs et opérateurs logiques

Les comparateurs et les opérateurs logiques fonctionnent de la même manière que les opérateurs arithmétiques. Toutefois, le résultat empilé sera soit 0 (faux), soit 1 (vrai). Les opérateurs logiques considèrent que 0 correspond à faux et une valeur différente de 0 à vrai (comme en C).

`= ( a b -- r , r vaut 1 si a est égal à b, 0 sinon )`
`< ( a b -- r , r vaut 1 si a est inférieur à b, 0 sinon )`
`<= ( a b -- r , r vaut 1 si a est inférieur ou égal à b, 0 sinon )`
`> ( a b -- r , r vaut 1 si a est superieur à b, 0 sinon )`
`>= ( a b -- r , r vaut 1 si a est supérieur ou égal à b, 0 sinon )`
`AND ( a b -- r , r vaut 1 si a et b sont différents de 0, 0 sinon )`
`OR ( a b -- r , r vaut 1 si a ou b sont différents de 0, 0 sinon )`
`NOT ( n -- r , r vaut 0 si n est différent de 0, 1 sinon )`

4.3.4 Mots de contrôle

Sélection

IF␣<*inst1*>␣ELSE␣<*inst2*>␣ENDIF

Le mot `IF` extrait le nombre du sommet de la pile. Si ce nombre est différent de 0, les mots *<inst1>* sont exécutés puis on saute après le `ENDIF`, sinon on saute les mots *<inst1>* et on exécute les mots *<inst2>*. Dans tous les cas, l'exécution reprend normalement après le `ENDIF`.

Le mot `else` étant facultatif, on dispose de la forme simplifiée de sélection suivante :

`IF␣`*<inst>*`␣ENDIF`

Exemples :

```
1␣2␣=␣if␣1␣else␣2␣endif␣.
```

provoque l'affichage de :

```
2
```

```
1␣2␣>␣if␣1␣.␣endif␣2␣.
```

provoque l'affichage de :

```
2
```

Répétition

`BEGIN␣`*<inst1>*`␣WHILE␣`*<inst2>*`␣REPEAT`

Le mot `WHILE` extrait le nombre du sommet de la pile. Si ce nombre est différent de 0, les mots *<inst2>* sont exécutés puis, en arrivant au mot `REPEAT`, on retourne au mot qui suit le `BEGIN` (partie *<inst1>*) ; sinon, on saute les mots *<inst2>* et l'exécution reprend normalement après le `REPEAT`.

Exemple :

```
1
begin
␣␣dup␣4␣<
while
␣␣dup␣1␣+
repeat
.s
```

provoque l'affichage de :

```
<4>␣1␣2␣3␣4␣
```

4.4 Définition de nouveaux mots

`:␣`*<mot>*`␣`*<corps>*`␣;`

Le mot `:` permet de définir un nouveau mot *<mot>*. La définition s'arrête au mot `;`. Lors de sa définition (qui peut être écrite n'importe où dans le fichier source), le nouveau mot est rajouté à un dictionnaire, mais les mots composant le *<corps>* de la définition ne sont pas exécutés. On suppose que le programme ne comporte pas de définitions imbriquées.

Après avoir été défini, un mot peut être utilisé. Lors de cette utilisation, ce sont les mots qui forment le *<corps>* de sa définition qui sont exécutés.

Les paramètres et les résultats sont obligatoirement passés par la pile de données.

Exemples :

```
:␣moyenne␣+␣2␣/␣;
```

permet de définir le mot `moyenne` qui calcule la moyenne (entière) des deux entiers placés au sommet de la pile. Ces entiers sont dépilés et le résultat est placé au sommet de la pile de données. Si la pile est initialement vide,
`8␣15␣moyenne␣.s`
provoque l'affichage de :
`<1>␣11␣`

```
:␣somme␣(␣n␣--␣somme[n]␣,␣somme␣des␣n␣premiers␣entiers␣)
0␣(␣valeur␣initiale␣de␣la␣somme␣)
begin
␣␣over␣0␣>␣(␣n>0␣?␣)
while
␣␣over␣+␣(␣ajoute␣n␣à␣la␣somme␣)
␣␣swap␣1␣-␣swap␣(␣décrémente␣n␣)
repeat
swap␣drop␣(␣enlève␣n␣de␣la␣pile␣)
;
4␣somme␣.
```

provoque l'affichage de :
`10`

Remarque : dans un programme YoctoForth, un commentaire commence au mot `(` et se termine au caractère `)`. Par exemple [1] :
`(␣Ceci␣est␣un␣commentaire␣)␣(␣et␣ceci␣aussi)`

Affichage des mots du dictionnaire :
`WORDS ( --  , affiche la liste des mots du dictionnaire )`
Par exemple, si l'on exécute le mot `words` après les deux définitions précédentes, on obtiendra l'affichage suivant :
`moyenne␣somme␣`

4.5 Entrées-sorties

`EMIT ( car -- , affiche le caractère dont le code est au sommet de la pile )`
`KEY ( -- car, empile le code du caractère tapé au clavier )`
`CHAR ( -- car, empile le code du caractère qui suit le mot CHAR )`
`CR ( -- , affiche le caractère « nouvelle ligne » )`
`SPACE ( -- , affiche le caractère espace )`
`SPACES ( n -- , si n > 0 affiche n espaces )`
`."  ( -- , affiche les caractères qui suivent jusqu'au premier " rencontré )`
`#IN ( -- n , empile l'entier tapé au clavier )`

Remarque : on suppose que pour le mot `."`, la chaîne à afficher n'excède pas 511 caractères.
Exemples [2] *:*

1. Attention aux espaces autour de la parenthèse ouvrante. C'est un mot, il faut donc qu'elle soit entourée de séparateurs. Ce n'est pas obligatoire pour la parenthèse fermante.
2. Le symbole ↩ représente le caractère « nouvelle ligne ».

Les mots :	provoquent l'affichage de :
97␣emit	a
key␣.	97 si l'utilisateur a tapé : a↩
char␣a␣.	97
cr	↩
space	␣
4␣spaces	␣␣␣␣
."␣␣Bonjour␣à␣tous␣!␣␣"	␣Bonjour␣à␣tous␣!␣␣
#in␣.	3172 si l'utilisateur a tapé : 3172↩

5 Travail à réaliser

5.1 Allocations dynamiques

Afin de vérifier le bon déroulement des allocations dynamiques de la mémoire, on n'utilisera pas les fonctions `malloc` et `free`, mais les fonctions suivantes disponibles dans le module `memoire` :

```
void *Allouer(size_t NbOctets)
void Liberer(void *Adresse)
void Bilan(void)
```

Ce module doit être récupéré sur Moodle. Le rôle de ces fonctions est décrit dans le fichier d'en-tête du module (`memoire.h`).

5.2 Lecture du fichier source

Il s'agit d'écrire un module[3] `lecture` permettant de lire un fichier source et de le stocker en mémoire dans la structure de données suivante :

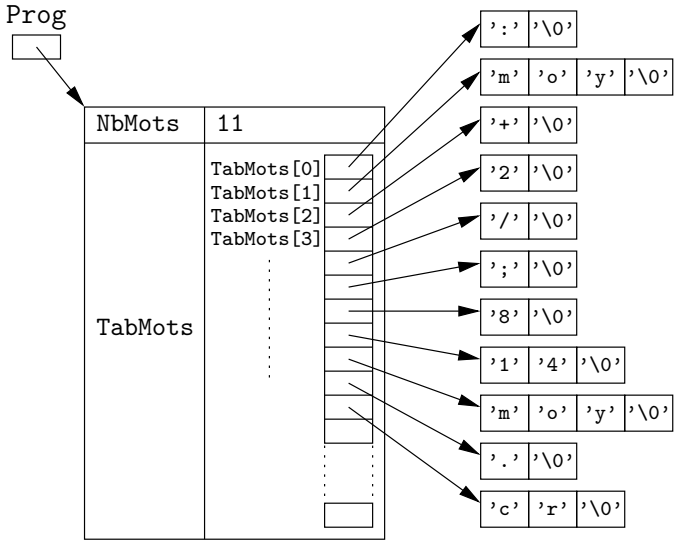
```
struct sProgramme
{
  int NbMots;
  char *TabMots[NB_MOTS_MAX];
};

typedef struct sProgramme *Programme;
```

Exemple : à l'issue de la lecture du fichier source contenant les mots suivants :

```
: MOY ( a b -- r , moyenne de a et b )
+ 2 / ;
8 14 moy . cr
```

la représentation mémoire du programme devra être :

3. Pour chaque module que vous devez écrire, vous devez appliquer les principes vus en CTD concernant la programmation modulaire.



Remarques :

— la lecture doit ignorer les commentaires ;

— majuscules et minuscules n'étant pas différenciées, toutes les lettres sont transformées en minuscules [4] ;

— on utilisera les deux constantes symboliques suivantes :

```
#define TAILLE_MOT_MAX 31
#define NB_MOTS_MAX 2048
```

qui représentent respectivement la longueur maximale d'un mot [5] et le nombre maximum de mots présents dans un programme.

Le module `lecture` devra comporter les fonctions suivantes :

- `Programme LectureProg(char NomFichier[])` : retourne le programme lu dans `NomFichier` ;
- `int LongueurProg(Programme Prog)` : retourne le nombre de mots contenus dans `Prog` ;
- `char *MotProg(Programme Prog, int i)` : retourne l'adresse du `i`e mot de `Prog` ;
- `void AfficherProg(Programme Prog)` : affiche sur `stdout` tous les mots de `Prog` [6] ;
- `void LibererProg(Programme Prog)` : libère la mémoire occupée par le programme accessible par `Prog`.

5.3 Gestion d'une pile d'entiers

Il s'agit d'écrire un module `pile` permettant de gérer une pile d'entiers qui sera représentée par la structure de données suivante :

```
#include <stdbool.h>

struct sCellule
{
  int Valeur;
  struct sCellule *Suivant;
};
```

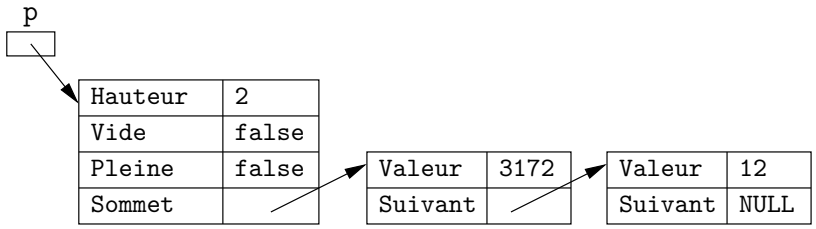
4. La fonction `tolower` (`ctype.h`) peut être utilisée.

5. Il y a deux exceptions : la chaîne qui suit le mot `."` (§ 4.5) et le nom du fichier qui suit le mot `INCLUDE` (§ 5.7) qui n'est pas stocké dans le programme.

6. Cette fonction sera surtout utilisée en phase de mise au point.

```
struct sPile
{
  unsigned int Hauteur;
  bool Vide;
  bool Pleine;
  struct sCellule *Sommet;
};

typedef struct sPile *Pile;
```

Remarque : l'inclusion du fichier `stdbool.h` permet de bénéficier de la définition du synonyme de type `bool` dont les variables peuvent prendre pour valeur l'une des deux constantes symboliques `true` et `false`.
Exemple : si les entiers 12 puis 3172 ont été empilés, la représentation mémoire devra être :



Le module `pile` devra comporter les fonctions suivantes :

- `Pile CreerPile(void)` : crée une pile vide ;
- `bool PileVide(Pile p)` : indique si `p` est vide ;
- `bool PilePleine(Pile p)` : indique si `p` est pleine ;
- `unsigned int HauteurPile(Pile p)` : retourne le nombre d'éléments de `p` ;
- `void Empiler(Pile p, int Elt)` : empile `Elt` sur `p` ; si `Hauteur` atteint la valeur de la constante symbolique `HAUTEUR_MAX` (fixée à 1024), affecte `true` à l'indicateur `Pleine` ;
- `int SommetPile(Pile p)` : retourne l'entier au sommet de `p` ou 0 si la pile était vide ;
- `int Depiler(Pile p)` : retourne et enlève l'entier du sommet de `p` ; si la pile était vide, 0 est retourné ; si la pile devient vide, `true` est affecté à l'indicateur `Vide` ;
- `void AfficherPile(Pile p)` : affiche le contenu de `p` au format décrit au paragraphe 4.3.1 pour le mot `.S` ;
- `void LibererPile(Pile p)` : libère la mémoire occupée par la pile accessible par `p`.

5.4 Interpréteur de niveau 1

Il s'agit d'écrire le fichier principal du projet qui va utiliser les modules précédents afin de réaliser un interpréteur minimal. Après avoir lu le fichier source à traiter dont le nom sera passé en paramètre, le cœur de l'interpréteur sera constitué d'une simple boucle qui peut se résumer ainsi :

Tant qu'il reste des mots à traiter faire :
— si le mot courant est une opération arithmétique (§ 4.3.2), un comparateur, un opérateur logique (§ 4.3.3) ou une opération de manipulation de la pile de données (§ 4.3.1), cette opération est effectuée ;
— sinon, si le mot courant est un entier [7], il est empilé sur la pile de données ;

7. La fonction `isdigit` (`ctype.h`) peut être utilisée.

— sinon, afficher un message précisant que le mot est invalide et interrompre l'exécution de l'interpréteur ;
— passer au mot suivant, c'est-à-dire incrémenter l'indice du mot à traiter dans le tableau `TabMots` (champ de `struct sProgramme`).

Dans cette version, on fera les vérifications portant sur les points suivants :
— la présence du nom du fichier source sur la ligne de commande ;
— l'existence du fichier source ;
— l'existence des mots : si le mot courant ne fait pas partie des groupes cités ci-dessus, un message devra être affiché sur `stderr` et l'exécution devra se terminer ;
— si la pile est vide lors d'une tentative de dépilement ou si elle est pleine lors d'une tentative d'empilement, un message devra être affiché sur `stderr` et l'exécution devra se terminer ;
— l'interpréteur ne doit pas avoir d'erreur lors de son exécution, quelles que soient les erreurs commises par l'utilisateur.

En ce qui concerne la lecture du fichier source, on supposera qu'il ne contient pas de commentaires.

Enfin, dès ce premier niveau, la compilation séparée devra être effectuée grâce à l'utilitaire `make`. L'écriture d'un fichier de description des dépendances `Makefile` est donc nécessaire.

Remarque : dans tous les modules, il est fortement conseillé d'écrire des fonctions « auxiliaires » qui ne seront pas publiques ; elles devront donc être de classe statique.

5.5 Interpréteur de niveau 2

Il s'agit de compléter l'interpréteur de niveau 1 en ajoutant les fonctionnalités suivantes :
— modification du module `lecture` afin de prendre en compte la présence d'éventuels commentaires (on supposera ici qu'ils sont correctement utilisés, c'est-à-dire qu'il n'y a pas de commentaires imbriqués et que les parenthèses sont en correspondance) ;
— modification du fichier principal afin de prendre en compte les mots d'entrées-sorties (§ 4.5) ; attention, le module `lecture` doit aussi être modifié afin de stocker dans un seul mot la chaîne de caractère qui doit être affichée grâce au mot `."` ;
— modification du fichier principal afin de prendre en compte les mots de contrôle (§ 4.3.4) ; on ne prendra pas en compte ici les structures de contrôle imbriquées ; toutefois, s'il y en a, le résultat de l'interprétation sera erroné mais elles ne doivent pas provoquer d'erreur à l'exécution de l'interpréteur, c'est-à-dire d'arrêt anormal de l'exécution de votre programme par le système d'exploitation, accompagné, par exemple, du message « Segmentation fault: 11 ».

5.6 Interpréteur de niveau 3

Il s'agit de compléter l'interpréteur de niveau 2 en ajoutant les fonctionnalités décrites ci-dessous.

5.6.1 Gestion d'un dictionnaire

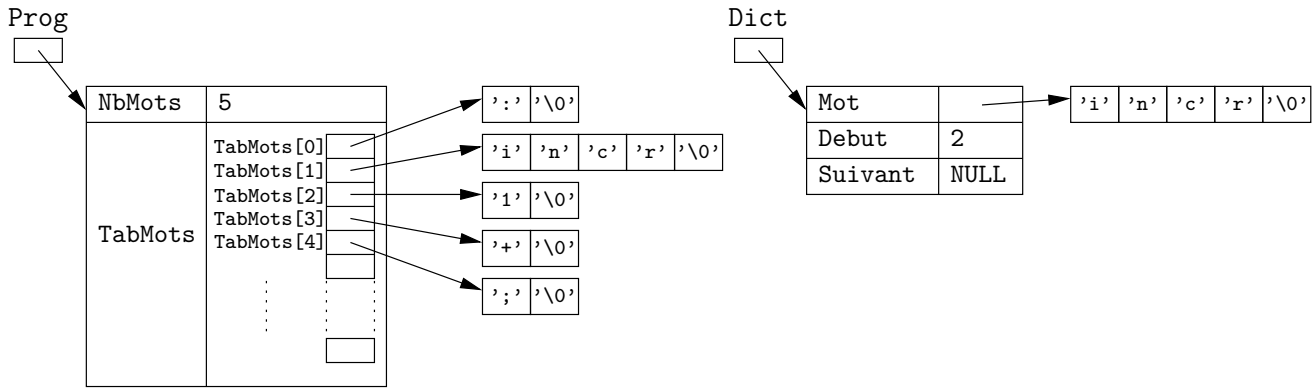
Il s'agit d'écrire un module `dico` permettant de gérer un dictionnaire qui sera représenté par la structure de données suivante :

```
struct sDictionnaire
{
  char *Mot;
  int Debut;
  struct sDictionnaire *Suivant;
};

typedef struct sDictionnaire *Dictionnaire;
```

Il s'agit donc d'une liste simplement chaînée basique. Chaque cellule contient un champ `Mot` qui désigne un nouveau mot défini par l'utilisateur et un champ `Debut` qui contient l'indice dans le tableau `TabMots` du premier mot du corps de la définition du nouveau mot en question.
Exemple : si le fichier source commence par les mots suivants :

```
: incr 1 + ;
```

la représentation mémoire du programme et du dictionnaire devra être :



Le champ `Debut` (ici 2) est égal à l'indice du premier mot du corps de la définition (ici le mot `1`) dans le tableau `TabMots`.

Le module `dico` devra comporter les fonctions suivantes :

- `Dictionnaire CreerDico(void)` : crée un dictionnaire vide ;
- `int AjouterDico(char *Mot, Dictionnaire *pDict, int Deb)` : ajoute le mot désigné par `Mot`, associé à l'indice de début `Deb`, au dictionnaire pointé par `pDict` ; la valeur retournée sera 1 si le mot était absent du dictionnaire, -1 si l'allocation mémoire a échoué ou 0 si le mot était déjà présent ; dans ce dernier cas, aucune nouvelle cellule n'est allouée, mais le champ `Debut` de la cellule existante doit être mis à jour ;
- `void AfficherDico(Dictionnaire Dict)` : affiche le contenu de `Dict` (voir la description de `WORDS` au paragraphe 4.4) ;
- `int RechercherDico(char *Mot, Dictionnaire Dict)` : retourne la valeur du champ `Debut` correspondant à `Mot` dans `Dict` ou -1 si le mot n'est pas présent ;
- `void LibererDico(Dictionnaire Dict)` : libère la mémoire occupée par le dictionnaire désigné par `Dict`.

5.6.2 Définition et utilisation de nouveaux mots

Afin de permettre l'utilisation de nouveaux mots, il faut ajouter une seconde pile d'entiers appelée *pile de retour*.

Définition. Lorsque l'interpréteur rencontre le mot `:`, le mot qui suit (`incr` dans l'exemple) ainsi que le numéro du mot suivant (2 dans l'exemple) sont rangés dans le dictionnaire. Si le nouveau mot est déjà présent dans le dictionnaire, un message avertissant de sa redéfinition doit être affiché. L'éxécution reprend alors au mot qui suit le `;`, c'est-à-dire après la définition (il faut prendre garde qu'une erreur de syntaxe YoctoForth ne provoque pas une erreur à l'exécution de l'interpréteur, c'est-à-dire un arrêt anormal de l'exécution de votre programme par le système d'exploitation, accompagné, par exemple, du message « Segmentation fault: 11 »).

Utilisation. Dans la boucle de l'interpréteur, il faut *avant toute chose* rechercher le mot courant dans le dictionnaire. S'il est présent, le numéro du mot suivant (l'indice de retour) est empilé sur la pile de retour et, au lieu de passer au mot suivant, l'indice du mot à traiter dans le tableau `TabMots` reçoit l'indice du début (champ `Debut`) de la définition du mot dans le dictionnaire Quand l'exécution est terminée (on arrive au `;`) dans la définition du mot, le numéro placé au sommet de la pile de retour est dépilé et affecté à l'indice du mot à traiter dans le tableau `TabMots`.

L'exécution reprend alors à l'endroit où elle avait été interrompue.
Exemple : soit le programme suivant :

```
: incr 1 + ;
3172 incr .
```

L'indice du mot à traiter dans le tableau `TabMots` est initialisé à 0. Le mot courant est donc `:`. L'interpréteur ajoute dans le dictionnaire le mot `incr` et le numéro du mot suivant, c'est-à-dire 2.

L'exécution reprend alors au mot qui suit le mot `;`, c'est-à-dire au mot `3172` (l'indice du mot à traiter dans le tableau `TabMots` vaut donc 5). La valeur 3172 est alors empilée sur la pile de donnée et on passe au mot suivant (l'indice du mot à traiter dans le tableau `TabMots` est incrémenté).

L'interpréteur constate que le mot `incr` est présent dans le dictionnaire. Le numéro du mot suivant (le mot `.`), c'est-à-dire 7, est empilé sur la pile de retour. L'indice du mot à traiter dans le tableau `TabMots` reçoit alors la valeur 2 qui est l'indice du début de la définition du mot `incr` dans le dictionnaire. L'exécution continue alors jusqu'au mot `;` (la valeur 1 est empilée sur la pile de donnée, les valeurs 1 et 3172 sont dépilées et la valeur 3173 est empilée). La valeur 7 est alors dépilée de la pile de retour et affectée à l'indice du mot à traiter dans le tableau `TabMots`. L'exécution reprend alors au mot `.` qui provoque le dépilement et l'affichage de la valeur 3173.

On ajoutera aussi à ce niveau la prise en compte du mot `WORDS`.

5.6.3 Structures de contrôle imbriquées

Il s'agit de prendre en compte la possibilité d'utiliser des sélections et des répétitions imbriquées. Pour les sélections, un compteur permet de résoudre le problème, alors que pour les répétitions, en plus d'un compteur, on utilisera la pile de retour qui permettra de stocker les différents « indices de rebouclage ».

5.7 Interpréteur de niveau 4

Il s'agit de compléter l'interpréteur de niveau 3 en ajoutant les fonctionnalités suivantes au module `lecture` :

- vérifications syntaxiques à la lecture du fichier (parenthèses pour les commentaires, guillemets pour le mot `."`, couple `: ;` et structures de contrôle) ; on pourra utiliser une pile spécifique ;
- inclusions de fichiers sources ; on ajoutera la prise en compte du mot suivant :
  `INCLUDE ( -- , inclut le contenu du fichier dont le nom est le mot suivant )`
  On suppose que les noms des fichiers ne contiennent pas d'espace. Un nom de fichier comporte au plus `FILENAME_MAX-1` caractères, la constante symbolique `FILENAME_MAX` étant définie dans le fichier `stdio.h`.

À ce niveau, toutes les améliorations seront les bienvenues.